

PROGRAMMING THE Z80



PROGRAMMING THE Z80

RODNAY ZAKS

THIRD EDITION



"Z80" is a registered trademark of ZILOG Inc., with whom SYBEX is not connected in any way.

Cover Design by Daniel le Noury

Every effort has been made to supply complete and accurate information. However, Sybex assumes no responsibility for its use; nor any infringements of patents or other rights of third parties which would result. No license is granted by the equipment manufacturers under any patent or patent rights. Manufacturers reserve the right to change circuitry at any time without notice.

In particular, technical characteristics and prices are subject to rapid change. Comparisons and evaluations are presented for their educational value and for guidance principles. The reader is referred to the manufacturer's data for exact specifications.

Copyright ©1980, SYBEX Inc. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to, photocopy, photograph, or magnetic or other record, without the prior written permission of the publisher.

Library of Congress Card Number: 80-5468

ISBN: 0-89588-094-6

First Edition published 1979. Third Edition 1981

Printed in the United States of America

Printing 10 9 8 7 6 5 4 3

ACKNOWLEDGEMENTS

Designing a programming textbook is always difficult. Designing it so that it will teach elementary programming as well as advanced concepts while covering both hardware and software aspects makes it a challenge. The author would like to acknowledge here the many constructive suggestions for improvements or changes made by: O.M. Barlow, Dennis L. Feick, Richard D. Reid, Stanley E. Erwin, Philip Hooper, Dennis B. Kitz, R. Ratke, and Jim Crocker.

A special acknowledgement is also due to Chris Williams for his contribution to the instruction-set and the data structures section.

Any additional suggestions for improvements or changes should be sent to the author, and will be reflected in forthcoming editions.

Several tables in Chapter Four showing hexadecimal codes for the Z80 instructions have been reprinted by permission of Zilog Inc. Tables 2.26 and 2.27 have been reprinted by permission of Intel Corporation.

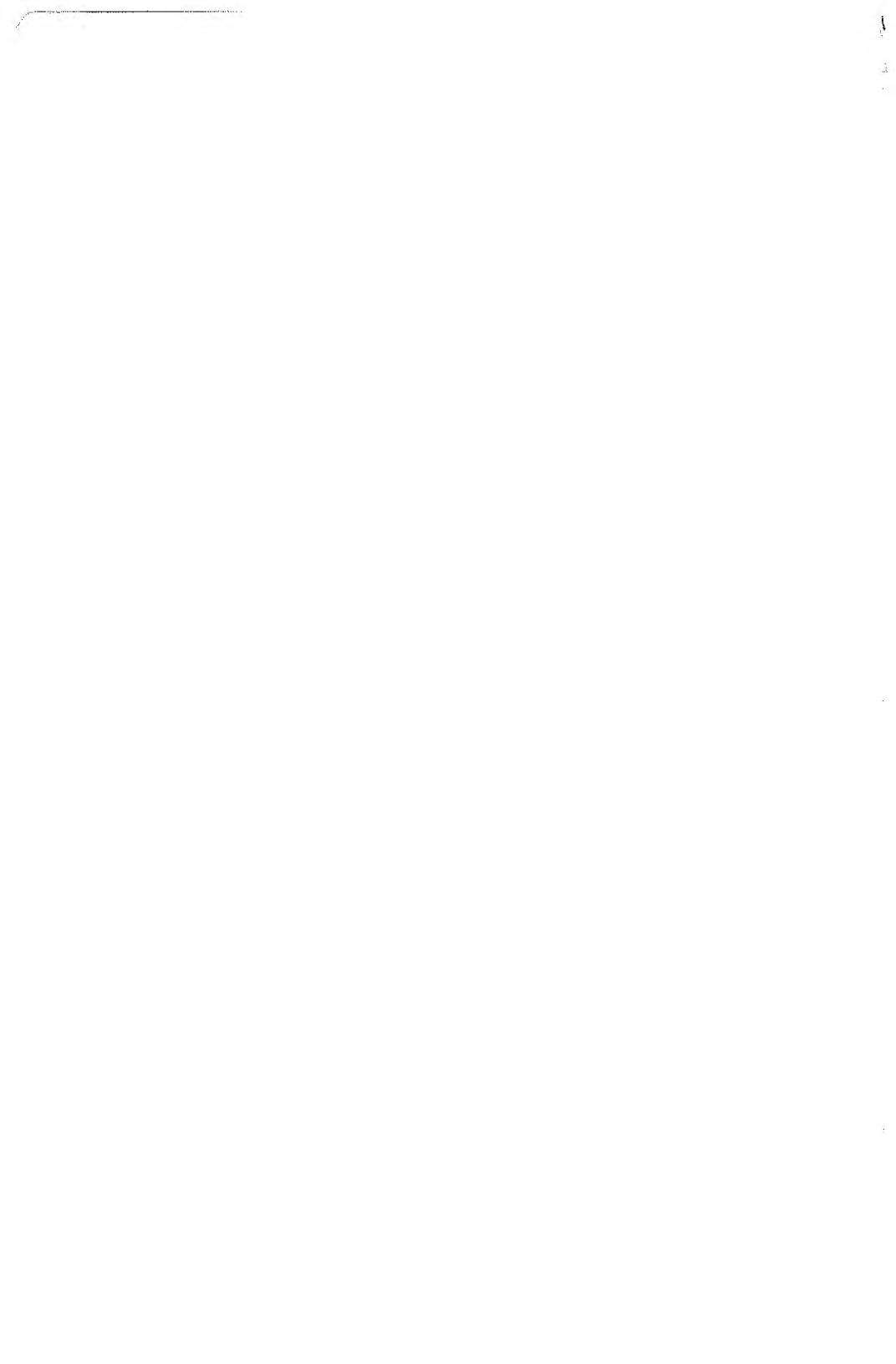


TABLE OF CONTENTS

PREFACE	13
I. BASIC CONCEPTS	15
<i>Introduction, What is programming?, Flowcharting, Information Representation</i>	
II. Z80 HARDWARE ORGANIZATION	46
<i>Introduction, System Architecture, Internal Organization of the Z80, Instruction Formats, Execution of Instructions with the Z80, Hardware Summary</i>	
III. BASIC PROGRAMMING TECHNIQUES	94
<i>Introduction, Arithmetic Programs, BCD Arithmetic Multiplication, Binary Division, Instruction Summary, Subroutines, Summary</i>	
IV. THE Z80 INSTRUCTION SET	154
<i>Introduction, Classes of Instructions, Summary, Individual Descriptions</i>	
V. ADDRESSING TECHNIQUES	438
<i>Introduction, Possible Addressing Modes, Z80 Addressing Modes, Using the Z80 Addressing Modes, Summary</i>	

VI. INPUT/OUTPUT TECHNIQUES **460**

Introduction, Input/output, Parallel Word Transfer, Bit Serial Transfer, Peripheral Summary, Input/Output Scheduling, Summary

VII. INPUT/OUTPUT DEVICES **511**

Introduction, The Standard PIO, The Internal Control Register, Programming a PIO, The Zilog Z80 PIO

VIII. APPLICATION EXAMPLES **520**

Introduction, Clearing a Section of Memory, Polling I/O Devices, Getting Characters In, Testing A Character, Bracket Testing, Parity Generation, Code Conversion: ASCII to BCD, Convert Hex to ASCII, Finding the Largest Element of a Table, Sum of N Elements, A Checksum Computation, Count the Zeroes, Block Transfer, BCD Block Transfer, Compare Two Signed 16-bit Numbers, Bubble-Sort, Summary

IX. DATA STRUCTURES **539**

PART 1—THEORY

Introduction, Pointers, Lists, Searching and Sorting, Section Summary

PART 2—DESIGN EXAMPLES

Introduction, Data Representation for the List, A Simple List, Alphabetic Set, Linked List, Summary

X. PROGRAM DEVELOPMENT **579**

Introduction, Basic Programming Choices, Software Support, The

XI. CONCLUSION	602
<i>Technological Development, The Next Step</i>	
APPENDIX A	604
<i>Hexadecimal Conversion Table</i>	
APPENDIX B	605
<i>ASCII Conversion Table</i>	
APPENDIX C	606
<i>Relative Branch Tables</i>	
APPENDIX D	607
<i>Decimal to BCD Conversion</i>	
APPENDIX E	608
<i>Z80 Instruction Codes</i>	
APPENDIX F	615
<i>Z80 to 8080 Equivalence</i>	
APPENDIX G	616
<i>8080 to Z80 Equivalence</i>	
INDEX	617

PREFACE

This book has been designed as a complete self-contained text for learning programming, using the Z80. It can be used by a person who has never programmed before, and should also be of value to anyone using the Z80.

For the person who has already programmed, this book will teach specific programming techniques using (or working around) the specific characteristics of the Z80. This text covers the elementary to intermediate techniques required to start programming effectively.

This text aims at providing a true level of competence to the person who wishes to program using this microprocessor. Naturally, no book will effectively teach how to program, unless one actually practices. However, it is hoped that this book will take the reader to the point where he feels that he can start programming by himself and can solve simple or even moderately complex problems using a microcomputer.

This book is based on the author's experience in teaching more than 1000 persons how to program microcomputers. As a result, it is strongly structured. Chapters normally go from the simple to the complex. For readers who have already learned elementary programming, the introductory chapter may be skipped. For others who have never programmed, the final sections of some chapters may require a second reading. The book has been designed to take the reader systematically through all the basic concepts and techniques required to build increasingly complex programs. It is, therefore, strongly suggested that the ordering of the chapters be followed. In addition, for effective results, it is important that the reader attempt to solve as many exercises as possible. The difficulty within the exercises has been carefully graduated. They are designed to verify that the material which has been presented is really understood. Without doing the programming exercises, it will not be possible to realize the full value of this book as an educational medium. Several of the exercises may require time, such as the multiplication exercise. However, by doing them, you will actually program and *learn by doing*. This is indispensable.

For those who have acquired a taste for programming when reaching the end of this volume, a companion volume is planned: the *Z80 Applications Book*.

Other books in this series cover programming for other popular microprocessors.

For those who wish to develop their hardware knowledge, it is suggested that the reference books *From Chips to Systems: an Introduction to Microprocessors* (ref. C201A) and *Microprocessor Interfacing Techniques* (ref. C207) be consulted.

The contents of this book have been checked carefully and are believed to be reliable. However, inevitably, some typographical or other errors will be found. The author will be grateful for any comments by alert readers so that future editions may benefit from their experience. Any other suggestions for improvements, such as other programs desired, developed, or found of value by readers, will be appreciated.

1

BASIC CONCEPTS

INTRODUCTION

This chapter will introduce the basic concepts and definitions relating to computer programming. The reader already familiar with these concepts may want to glance quickly at the contents of this chapter and then move on to Chapter 2. It is suggested, however, that even the experienced reader look at the contents of this introductory chapter. Many significant concepts are presented here including, for example, two's complement, BCD, and other representations. Some of these concepts may be new to the reader; others may improve the knowledge and skills of experienced programmers.

WHAT IS PROGRAMMING?

Given a problem, one must first devise a solution. This solution, expressed as a step-by-step procedure, is called an *algorithm*. An algorithm is a step-by-step specification of the solution to a given problem. It must terminate in a finite number of steps. This algorithm may be expressed in any language or symbolism. A simple example of an algorithm is:

- 1—insert key in the keyhole
- 2—turn key one full turn to the left
- 3—seize doorknob
- 4—turn doorknob left and push the door

At this point, if the algorithm is correct for the type of lock involved, the door will open. This four-step procedure qualifies as an algorithm for door opening.

Once a solution to a problem has been expressed in the form of an algorithm, the algorithm must be executed by the computer. Unfortunately, it is now a well-established fact that computers cannot understand or execute ordinary spoken English (or any other human language). The reason lies in the *syntactic ambiguity* of all common human languages. Only a well-defined subset of natural language can be "understood" by the computer. This is called a *programming language*.

Converting an algorithm into a sequence of instructions in a programming language is called *programming*. To be more specific, the actual translation phase of the algorithm into the programming language is called *coding*. Programming really refers not just to the coding but also to the overall design of the programs and "data structures" which will implement the algorithm.

Effective programming requires not only understanding the possible implementation techniques for standard algorithms, but also the skillful use of all the computer hardware resources, such as internal registers, memory, and peripheral devices, plus a creative use of appropriate data structures. These techniques will be covered in the next chapters.

Programming also requires a strict documentation discipline, so that the programs are understandable to others, as well as to the author. Documentation must be both internal and external to the program.

Internal program documentation refers to the comments placed in the body of a program, which explain its operation.

External documentation refers to the design documents which are separate from the program: written explanations, manuals, and flowcharts.

FLOWCHARTING

One intermediate step is almost always used between the *algorithm* and the *program*. It is called a *flowchart*. A flowchart is simply a symbolic representation of the algorithm expressed as a sequence of rectangles and diamonds containing the steps of the algorithm. Rectangles are used for *commands*, or "executable statements." Diamonds are used for *tests* such as: If information

X is true, then take action A, else B. Instead of presenting a formal definition of flowcharts at this point, we will introduce and discuss flowcharts later on in the book when we present programs.

Flowcharting is a highly recommended intermediate step between the algorithm specification and the actual coding of the solution. Remarkably, it has been observed that perhaps 10% of the programming population can write a program successfully without having to flowchart. Unfortunately, it has also been observed that 90% of the population believes it belongs to this 10%! The result: 80% of these programs, on the average, will fail the first time they are run on a computer. (These percentages are naturally not meant to be accurate.) In short, most novice programmers seldom see the necessity of drawing a flowchart. This usually results in "unclean" or erroneous programs. They must then spend a long time testing and correcting their program (this is called the

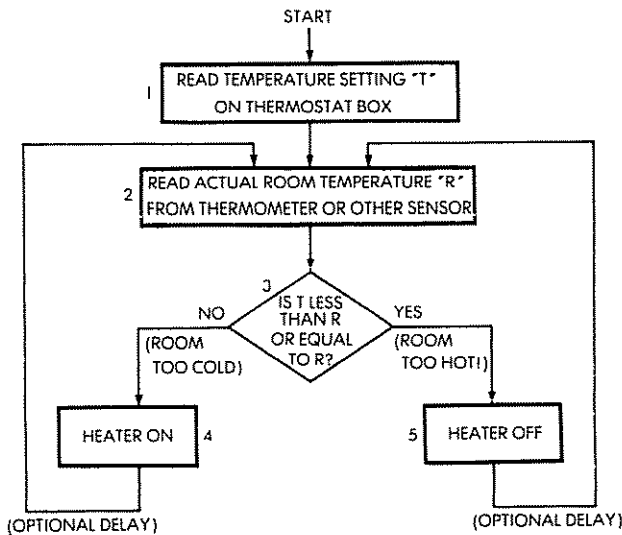


Fig. 1.1: A Flowchart for Keeping Room

debugging phase). The discipline of flowcharting is therefore highly recommended in all cases. It will require a small amount of additional time prior to the coding, but will usually result in a clear program which executes correctly and quickly. Once flowcharting is well understood, a small percentage of programmers will be able to perform this step mentally without having to do it on paper. Unfortunately, in such cases the programs that they write will usually be hard to understand for anybody else without the documentation provided by flowcharts. As a result, it is universally recommended that flowcharting be used as a strict discipline for any significant program. Many examples will be provided throughout the book.

INFORMATION REPRESENTATION

All computers manipulate information in the form of numbers or in the form of characters. Let us examine here the external and internal representations of information in a computer.

INTERNAL REPRESENTATION OF INFORMATION

All information in a computer is stored as groups of bits. A *bit* stands for a *binary digit* ("0" or "1"). Because of the limitations of conventional electronics, the only practical representation of information uses two-state logic (the representation of the state "0" and "1"). The two states of the circuits used in digital electronics are generally "on" or "off", and these are represented logically by the symbols "0" or "1". Because these circuits are used to implement "logical" functions, they are called "binary logic." As a result, virtually all information-processing today is performed in binary format. In the case of microprocessors in general, and of the Z80 in particular, these bits are structured in groups of eight. A group of eight bits is called a *byte*. A group of four bits is called a *nibble*.

Let us now examine how information is represented internally in this binary format. Two entities must be represented inside the computer. The first one is the program, which is a sequence of instructions. The second one is the data on which the program will operate, which may include numbers or alphanumeric text. We will discuss below three representations: program, numbers, and alphanumeric.

Program Representation

All instructions are represented internally as single or multiple bytes. A so-called "short instruction" is represented by a single byte. A longer instruction will be represented by two or more bytes. Because the Z80 is an eight-bit microprocessor, it fetches bytes successively from its memory. Therefore, a single-byte instruction always has a potential for executing faster than a two- or three-byte instruction. It will be seen later that this is an important feature of the instruction set of any microprocessor and in particular the Z80, where a special effort has been made to provide as many single-byte instructions as possible in order to improve the efficiency of the program execution. However, the limitation to 8 bits in length has resulted in important restrictions which will be outlined. This is a classic example of the compromise between speed and flexibility in programming. The binary code used to represent instructions is dictated by the manufacturer. The Z80, like any other microprocessor, comes equipped with a fixed instruction set. These instructions are defined by the manufacturer and are listed at the end of this book, with their code. Any program will be expressed as a sequence of these binary instructions. The Z80 instructions are presented in Chapter 4.

Representing Numeric Data

Representing numbers is not quite straightforward, and several cases must be distinguished. We must first represent integers, then signed numbers, i.e., positive and negative numbers, and finally we must be able to represent decimal numbers. Let us now address these requirements and possible solutions.

Representing integers may be performed by using a *direct binary* representation. The direct binary representation is simply the representation of the decimal value of a number in the binary system. In the binary system, the right-most bit represents 2 to the power 0. The next one to the left represents 2 to the power 1, the next represents 2 to the power 2, and the left-most bit represents 2 to the power 7 = 128.

$$\begin{array}{c}
 b_7b_6b_5b_4b_3b_2b_1b_0 \\
 \text{represents} \\
 b_72^7 + b_62^6 + b_52^5 + b_42^4 + b_32^3 + b_22^2 + b_12^1 + b_02^0
 \end{array}$$

The powers of 2 are:

$$2^7 = 128, 2^6 = 64, 2^5 = 32, 2^4 = 16, 2^3 = 8, 2^2 = 4, 2^1 = 2, 2^0 = 1$$

The binary representation is analogous to the decimal representation of numbers, where "123" represents:

$$\begin{array}{r} 1 \times 100 = 100 \\ + 2 \times 10 = 20 \\ + 3 \times 1 = 3 \\ \hline = 123 \end{array}$$

Note that $100 = 10^2$, $10 = 10^1$, $1 = 10^0$

In this "positional notation," each digit represents a power of 10. In the binary system, each binary digit or "bit" represents a power of 2, instead of a power of 10 in the decimal system.

Example: "00001001" in binary represents:

$$\begin{array}{r} 1 \times 1 = 1 \quad (2^0) \\ 0 \times 2 = 0 \quad (2^1) \\ 0 \times 4 = 0 \quad (2^2) \\ 1 \times 8 = 8 \quad (2^3) \\ 0 \times 16 = 0 \quad (2^4) \\ 0 \times 32 = 0 \quad (2^5) \\ 0 \times 64 = 0 \quad (2^6) \\ 0 \times 128 = 0 \quad (2^7) \\ \hline \end{array}$$

in decimal: $\quad = 9$

Let us examine some more examples:

By examining the binary representation of numbers, you will understand why bits are numbered from 0 to 7, going from right to left. Bit 0 is “b₀” and corresponds to 2⁰. Bit 1 is “b₁” and corresponds to 2¹, and so on.

Decimal	Binary	Decimal	Binary
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010	.	
3	00000011	.	
4	00000100	.	
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000	.	
9	00001001	.	
10	00001010	127	01111111
11	00001011	128	10000000
12	00001100	129	10000001
13	00001101		
14	00001110	.	
15	00001111	.	
16	00010000	.	
17	00010001	.	

Decimal to Binary

Conversely, let us compute the binary equivalent of "11" decimal:

$$\begin{array}{rcl}
 11 \div 2 = 5 \text{ remains } 1 & \rightarrow & 1 \quad \text{(LSB)} \\
 5 \div 2 = 2 \text{ remains } 1 & \rightarrow & 1 \\
 2 \div 2 = 1 \text{ remains } 0 & \rightarrow & 0 \\
 1 \div 2 = 0 \text{ remains } 1 & \rightarrow & 1 \quad \text{(MSB)}
 \end{array}$$

The binary equivalent is 1011 (read right-most column from bottom to top).

The binary equivalent of a decimal number may be obtained by dividing successively by 2 until a quotient of 0 is obtained.

Exercise 1.2: What is the binary for 257?

Exercise 1.3: Convert 19 to binary, then back to decimal.

Operating on Binary Data

The arithmetic rules for binary numbers are straightforward. The rules for addition are:

$$\begin{array}{rcl}
 0 + 0 & = & 0 \\
 0 + 1 & = & 1 \\
 1 + 0 & = & 1 \\
 1 + 1 & = & (1) \ 0
 \end{array}$$

where (1) denotes a "carry" of 1 (note that "10" is the binary equivalent of "2" decimal). Binary subtraction will be performed

Adding the next column:

$$\begin{array}{r} 10 \\ +01 \\ \hline 11 \end{array} \quad (1 + 0 = 1. \text{ No carry.})$$

Exercise 1.4: Compute $5 + 10$ in binary. Verify that the result is 15.

Some additional examples of binary addition:

$$\begin{array}{r} 0010 \quad (2) \\ +0001 \quad (1) \\ \hline =0011 \quad (3) \end{array}$$

$$\begin{array}{r} 0011 \quad (3) \\ +0001 \quad (1) \\ \hline =0100 \quad (4) \end{array}$$

This last example illustrates the role of the carry.

Looking at the right-most bits: $1 + 1 = (1) 0$

A carry of 1 is generated, which must be added to the next bits:

$$\begin{array}{r} 001 - \text{column 0 has just been added} \\ +000 - \\ + 1 \quad (\text{carry}) \\ \hline = (1)0 - \text{where (1) indicates a new} \\ \quad \quad \quad \text{carry into column 2.} \end{array}$$

The final result is: 0100

Another example:

Does the result hold in four bits?

With eight bits, it is therefore possible to represent directly the numbers "00000000" to "11111111," i.e., "0" to "255". Two obstacles should be visible immediately. First, we are only representing positive numbers. Second, the magnitude of these numbers is limited to 255 if we use only eight bits. Let us address each of these problems in turn.

Signed Binary

In a signed binary representation, the left-most bit is used to indicate the sign of the number. Traditionally, "0" is used to denote a *positive* number while "1" is used to denote a *negative* number. Now "11111111" will represent -127 , while "01111111" will represent $+127$. We can now represent positive and negative numbers, but we have reduced the maximum magnitude of these numbers to 127.

Example: "0000 0001" represents $+1$ (the leading "0" is "+", followed by "000 0001" = 1).

"1000 0001" is -1 (the leading "1" is "-").

Exercise 1.6: What is the representation of -5 in signed binary?

10000101

Let us now address the *magnitude* problem: in order to represent larger numbers, it will be necessary to use a larger number of bits. For example, if we use sixteen bits (two bytes) to represent numbers, we will be able to represent numbers from $-32K$ to $+32K$ in signed binary (1K in computer jargon represents 1,024). Bit 15 is used for the sign, and the remaining 15 bits (bit 14

binary representation which we have introduced. Let us add “-5” and “+7”.

+7 is represented by	00000111
-5 is represented by	10000101
<hr/>	
The binary sum is:	10001100, or -12

This is not the correct result. The correct result should be +2. In order to use this representation, special actions must be taken, depending on the sign. This results in increased complexity and reduced performance. In other words, the binary addition of signed numbers does not “work correctly.” This is annoying. Clearly, the computer must not only represent information, but also perform arithmetic on it.

The solution to this problem is called the *two's complement* representation, which will be used instead of the *signed binary* representation. In order to introduce two's complement let us first introduce an intermediate step: *one's complement*.

One's Complement

In the one's complement representation, all positive integers are represented in their correct binary format. For example “+3” is represented as usual by 00000011. However, its complement “-3” is obtained by complementing every bit in the original representation. Each 0 is transformed into a 1 and each 1 is transformed into a 0. In our example, the one's complement representation of “-3” will be 11111100.

Another example:

+2 is	00000010
-2 is	11111101

$$\begin{array}{r} -4 \text{ is } 11111011 \\ +6 \text{ is } 00000110 \\ \hline \end{array}$$

the sum is: (1) 00000001 where (1) indicates a carry

The "correct result" should be "2", or "00000010".

Let us try again:

$$\begin{array}{r} -3 \text{ is } 11111100 \\ -2 \text{ is } 11111101 \\ \hline \end{array}$$

The sum is: (1) 11111001

or "-6," plus a carry. The correct result should be "-." The representation of "-5" is 1111010. It did not work.

This representation does represent positive and negative numbers. However the result of an ordinary addition does not always come out "correctly." We will use still another representation. It is evolved from the one's complement and is called the two's complement representation.

Two's Complement Representation

In the two's complement representation, positive numbers are still represented, as usual, in signed binary, just like in one's complement. The difference lies in the representation of *negative numbers*. A negative number represented in two's complement is obtained by first computing the one's complement, and then *adding one*. Let us examine this in an example:

+3 is represented in signed binary by 00000011. Its one's complement representation is 11111100. The two's complement is obtained by adding one. It is 11111101.

Let us try a subtraction:

$$\begin{array}{r}
 (3) \quad 00000011 \\
 (-5) \quad +1111011 \\
 \hline
 =11111110
 \end{array}$$

Let us identify the result by computing the two's complement:

$$\begin{array}{r}
 \text{the one's complement of } 11111110 \text{ is } 00000001 \\
 \text{Adding 1} \quad + \quad 1 \\
 \hline
 \end{array}$$

therefore the two's complement is 00000010 or $+2$

Our result above, "11111110" represents "-2". It is correct.

We have now tried addition and subtraction, and the results were correct (ignoring the carry). It seems that two's complement works!

Exercise 1.8: What is the two's complement representation of "+127"?

$$01111111$$

Exercise 1.9: What is the two's complement representation of "-128"?

$$10000000$$

Let us now add $+4$ and -3 (the subtraction is performed by adding the two's complement):

From this point on, all signed integers will implicitly be represented internally in two's complement notation. See Fig. 1.3 for a table of two's complement numbers.

Exercise 1.10: What are the smallest and the largest numbers which one may represent in two's complement notation, using only one byte? $-128, +127$

Exercise 1.11: Compute the two's complement of 20. Then compute the two's complement of your result. Do you find 20 again?

The following examples will serve to demonstrate the rules of two's complement. In particular, C denotes a possible carry (or borrow) condition. (It is bit 8 of the result.)

V denotes a two's complement overflow

+	2's complement code	—	2's complement code
+127	01111111	—128	10000000
+126	01111110	—127	10000001
+125	01111101	—126	10000010
		—125	10000011
+65	01000001	—65	10111111
+64	01000000	—64	11000000
+63	00111111	—63	11000001
+33	00100001	—33	11011111
+32	00100000	—32	11100000
+31	00011111	—31	11100001
+17	00010001	—17	11101111
+16	00010000	—16	11110000
+15	00001111	—15	11110001
+14	00001110	—14	11110010
+13	00001101	—13	11110011
+12	00001100	—	

Overflow V

Here is an example of overflow:

$$\begin{array}{r}
 \begin{array}{l} \text{bit 6} \\ \text{bit 7} \end{array} \begin{array}{l} \text{---} \\ \text{---} \end{array} \downarrow \downarrow \\
 \begin{array}{r}
 01000000 \quad (64) \\
 + 01000001 \quad + (65) \\
 \hline
 = 10000001 \quad = (-127)
 \end{array}
 \end{array}$$

An internal carry has been generated from bit 6 into bit 7. This is called an overflow.

The result is now negative, "by accident." This situation must be detected, so that it can be corrected.

Let us examine another situation:

$$\begin{array}{r}
 \begin{array}{r}
 11111111 \quad (-1) \\
 + 11111111 \quad + (-1) \\
 \hline
 = (1) \quad 11111110 \quad = (-2) \\
 \downarrow \\
 \text{carry}
 \end{array}
 \end{array}$$

In this case, an internal carry has been generated from bit 6 into bit 7, and also from bit 7 into bit 8 (the formal "Carry" C we have examined in the preceding section). The rules of two's complement arithmetic specify that this carry should be ignored. The result is then correct.

This is because the carry from bit 6 into bit 7 did not change the sign bit.

Overflow will occur in four situations:

- 1—adding large positive numbers
- 2—adding large negative numbers
- 3—subtracting a large positive number from a large negative number
- 4—subtracting a large negative number from a large positive number.

Let us now improve our definition of the overflow:

Technically, the overflow indicator, a special bit reserved for this purpose, and called a "flag," will be set when there is a carry from bit 6 into bit 7 and no external carry, or else when there is no carry from bit 6 into bit 7 but there is an external carry. This indicates that bit 7, i.e., the sign of the result, has been accidentally changed. For the technically-minded reader, the overflow flag is set by Exclusive

PROGRAMMING THE Z80

Positive-Positive

```
      00000110  (+6)
+ 00001000  (+8)
-----
= 00001110  (+14)  V:0      C:0
```

(CORRECT)

Positive-Positive with Overflow

```
      01111111  (+127)
+ 00000001  (+1)
-----
= 10000000  (-128) V:1      C:0
```

The above is invalid because an overflow has occurred.

(ERROR)

Positive-Negative (result positive)

```
      00000100  (+4)
+ 11111110  (-2)
-----
=(1)00000010  (+2)  V:0      C:1 (disregard)
```

(CORRECT)

Positive-Negative (result negative)

```
      00000010  (+2)
+ 11111100  (-4)
-----
= 11111110  (-2)  V:0      C:0
```

(CORRECT)

Negative-Negative

```
      11111110  (-2)
+ 111111
```

This time an "underflow" has occurred, by adding two large negative numbers. The result would be -189 , which is too large to reside in eight bits.

Exercise 1.12: Complete the following additions. Indicate the result, the carry C , the overflow V , and whether the result is correct or not:

$$\begin{array}{r} 10111111 \\ +11000001 \\ \hline \end{array}$$

$=$ _____ V: _____ C: _____
☐ CORRECT ☐ ERROR

$$\begin{array}{$$

then be used. For example, let us examine a 16-bit, "double-precision" format:

00000000	00000000	is "0"
00000000	00000001	is "1"
01111111	11111111	is "32767"
11111111	11111111	is "-1"
11111111	11111110	is "-2"

plication will be shown in Chapter 4.

This fixed-format representation may cause a loss of precision, but it may be sufficient for usual computations or mathematical operations.

Unfortunately, in the case of accounting, no loss of precision is tolerable. For example, if a customer rings up a large total on a cash register, it would not be acceptable to have a five figure amount to pay, which would be approximated to the dollar. Another representation must be used wherever precision in the result is essential. The solution normally used is *BCD*, or binary-coded decimal.

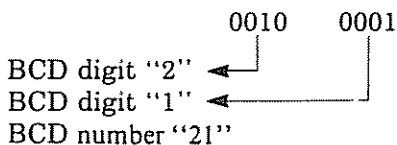
BCD Representation

The principle used in representing numbers in BCD is to encode each decimal digit separately,

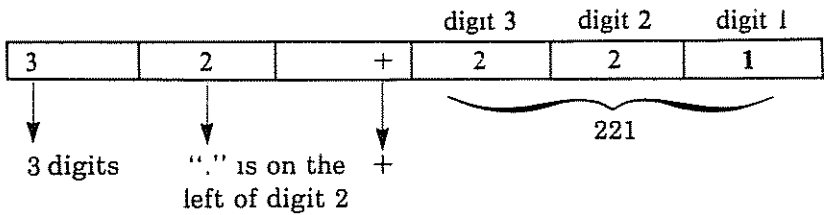
Since only four bits are needed to encode a BCD digit, two BCD digits may be encoded in every byte. This is called "*packed BCD*."

As an example, "00000000" will be "00" in BCD. "10011001" will be "99".

A BCD code is read as follows:



For example, +2.21 may be represented by:



It can be readily seen that a normalized number is characterized by a mantissa less than 1 and greater or equal to .1 in all cases where the number is not zero. In other words, this can be represented mathematically by:

$$.1 \leq M < 1 \text{ or } 10^{-1} \leq M < 10^0$$

Similarly, in the binary representation:

$$2^{-1} \leq M < 2^0 \text{ (or } .5 \leq M < 1)$$

Where M is the absolute value of the mantissa (disregarding the sign).

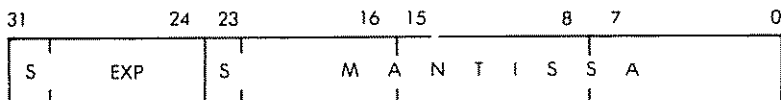
For example:

111.01 is normalized as: $.11101 \times 2^3$.

The mantissa is 11101.

The exponent is 3.

Now that we have defined the principle of the representation, let us examine the actual format. A typical floating-point representation appears below.



Exercise 1.21: How many decimal digits can the mantissa represent with the 23 bits?

This is only one example of a floating point representation. It is possible to use only three bytes, or it is possible to use more. The four-byte representation proposed above is just a common one which represents a reasonable compromise in terms of accuracy, magnitude of numbers, storage utilization, and efficiency in arithmetic operation.

We have now explored the problems associated with the representation of numbers and we know how to represent them in integer form, with a sign, or in decimal form. Let us now examine how to represent alphanumeric data internally.

Representing Alphanumeric Data

The representation of alphanumeric data, i.e. characters, is completely straightforward:

The table of 7-bit ASCII codes is shown in Fig. 1-6. In practice, it is used "as is," i.e. without parity, by adding a 0 in the left-most position, or else with parity, by adding the appropriate extra bit on the left.

Exercise 1.22: Compute the 8-bit representation of the digits "0" through "9", using even parity. (This code will be used in application examples of Chapter 8.)

Exercise 1.23: Same for the letters "A" through "F"

Exercise 1.24: Using a non-parity ASCII code (where the left-most bit is "0"), indicate the binary contents of the 4 characters below:

"A"
 "?"
 "3"
 "b"

HEX	MSD	0	1	2	3	4	5	6	7
LSB	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	—	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4									

We have examined the usual representations for both program and data inside the computer. Let us now examine the possible external representations.

EXTERNAL REPRESENTATION OF INFORMATION

The external representation refers to the way information is presented to the *user*, i.e. generally to the programmer. Information may be presented externally in essentially three formats: binary, octal or hexadecimal and symbolic.

1. Binary

It has been seen that information is stored internally in *bytes*, which are sequences of eight *bits* (0's or 1's). It is sometimes desirable to display this internal information directly in its binary format and this is called *binary representation*. One simple example is provided by Light

binary	octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Fig. 1.7: Octal Symbols

For example, "00 100 100" binary is represented by:

▼ ▼ ▼
 0 4 4

or "044" in octal.

Another example: 11 111 111 is:

▼ ▼ ▼
 3 7 7

or "377" in octal.

Conversely, the octal "211" represents:

010 001 001

or "10001001" binary.

Octal has traditionally been used on older computers which were employing various numbers of bits ranging from 8 to perhaps 64. More recently, with the dominance

DECIMAL	BINARY	HEX	OCTAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Fig. 1.8: Hexadecimal Codes

Example: $\underbrace{1010}_A \underbrace{0001}_1$ in binary is represented by
 A 1 in hexadecimal.

Exercise 1.25: What is the hexadecimal representation of "10101010"?

Exercise 1.26: Conversely, what is the binary equivalent of "FA" hexadecimal?

Exercise 1.27: What is the octal of "01000001"?

Hexadecimal offers the advantage of encoding eight bits into only two digits. This is easier to visualize or memorize and faster to type into a computer than its binary equivalent. Therefore, on most new microcomputers, hexadecimal

available on the less expensive systems. An alternative type of representation is then used, and in this case hexadecimal is the dominant representation. Only in rare cases relating to fine de-bugging at the hardware or the software level is the binary representation used. *Binary* directly displays the contents of registers of memory in binary format.

(The utility of a direct binary display on a front panel has always been the subject of a heated emotional controversy, which will not be debated here.)

We have seen how to represent information internally and externally. We will now examine the actual microprocessor which will manipulate this information.

Additional Exercises

Exercise 1.28: What is the advantage of two's complement over other representations used to represent signed numbers?

Exercise 1.29: How would you represent "1024" in direct binary? Signed binary? Two's complement?

Exercise 1.30: What is the V-bit? Should the programmer test it after an addition or subtraction?

Exercise 1.31: Compute the two's complement of "+16", "+17", "+18", "-16", "-17", "-18"

Exercise 1.32: Show the hexadecimal representation of the following text, which has been stored internally in ASCII format, with no parity: = "MESSAGE".

2

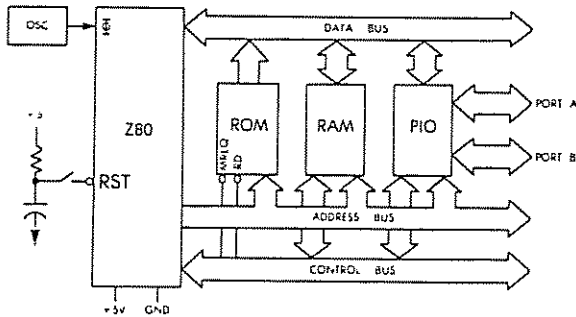
Z80 HARDWARE ORGANIZATION

INTRODUCTION

In order to program at an elementary level, it is not necessary to understand in detail the internal structure of the processor that one is using. However, in order to do efficient programming, such an understanding is required. The purpose of this chapter is to present the basic hardware concepts necessary for understanding the operation of the Z80 system. The complete microcomputer system includes not only the microprocessor unit (here the Z80), but also other components. This chapter presents the Z80 proper, while the other devices (mainly input/output) will be presented in a separate chapter (Chapter 7).

We will review here the basic architecture of the microcomputer system, then study more closely the internal organization of the Z80. We will examine, in particular, the various registers. We will then study the program execution and sequencing mechanism. From a hardware standpoint, this chapter is

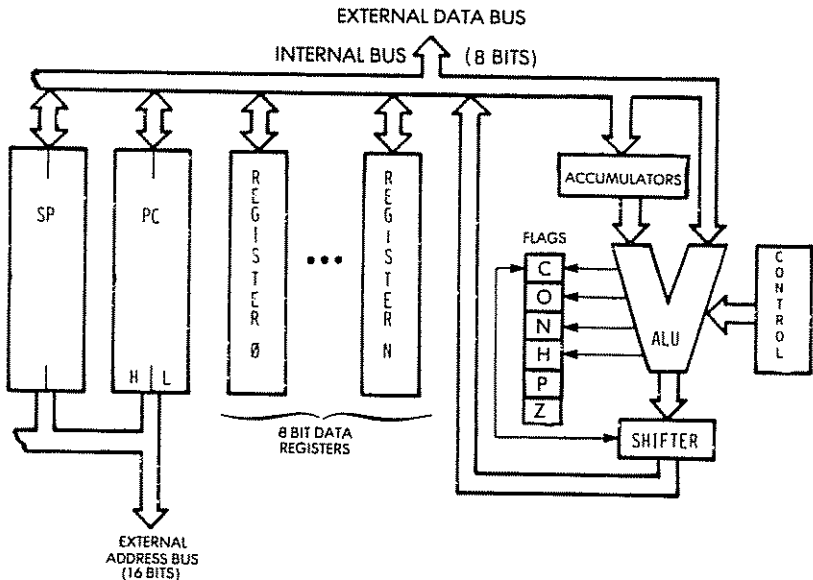
charge of sequencing the system. Its operation will be explained in this chapter.



ticular, the buses usually need to be *buffered*. Also, *decoding logic* may be used for the memory RAM chips, and, finally, some signals may need to be amplified by *drivers*. These auxiliary circuits will not be described here as they are not relevant to programming. The reader interested in specific assembly and interfacing techniques is referred to book C207 "Microprocessor Interfacing Techniques."

INSIDE A MICROPROCESSOR

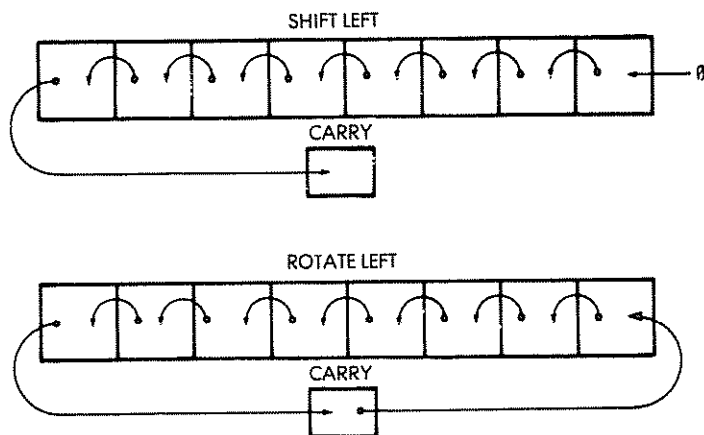
The large majority of all microprocessor chips on the market today implement the same architecture. This "standard" architecture will be described here. It is shown in Figure 2.2. The modules of this standard microprocessor will now be detailed, from right to left.



The *ALU* performs arithmetic and logic operations. A special register equips one of the inputs of the *ALU*, the left input here. It is called the accumulator. (Several accumulators may be provided.) The accumulator may be referenced both as input and output (source and destination) within the same instruction.

The *ALU* must also provide *shift* and *rotate* facilities.

A shift operation consists of moving the contents of a byte by one or more positions to the left or to the right. This is illustrated in Figure 2.3. Each bit has been moved to the left by one position. The details of shifts and rotations will be presented in the next chapter.



Setting Flags

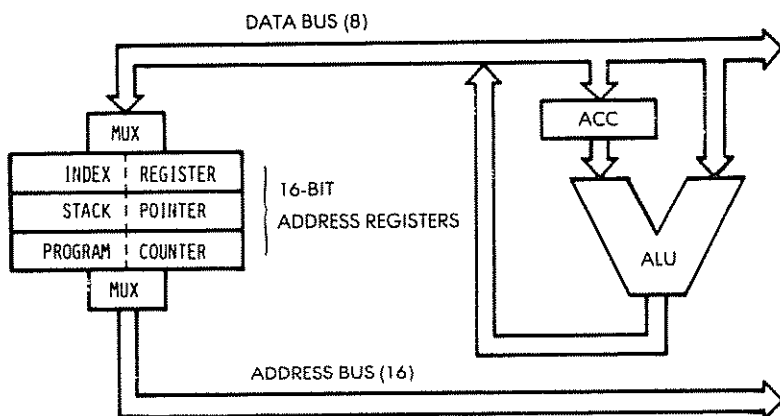
Most of the instructions executed by the processor will modify some or all of the flags. It is important to always refer to the chart provided by the manufacturer listing which bits will be modified by the instructions. This is essential in understanding the way a program is being executed. Such a chart for the Z80 is shown in Figure 4-17.

The Registers

Let us look now at Figure 2.2. On the left of the illustration, the registers of the microprocessor appear. Conceptually, one can distinguish the *general purpose registers* and the *address registers*.

The General-Purpose Registers

The only way to load the contents of these 16-bit registers is via the data bus. Two transfers will be necessary along the data bus in order to transfer 16 bits. In order to differentiate between the lower half and the higher half of each register, they are usually labelled as L (low) or H (high), denoting bits 0 through 7, and 8 through 15 respectively. This label is used whenever it is necessary to differentiate the halves of these registers. At least two address registers are present within most microprocessors. "MUX" in Fig. 2.4 stands for multiplexer. "MUX" in Fig. 2.4 stands for multiplexer.



In a few exceptional microprocessors, such as the two-chip F8, there is no PC on the microprocessor. This does not mean that the system does not have a program counter. The PC happens to be implemented directly on the memory chip, for reasons of efficiency.

Stack Pointer (SP)

The *stack* has not been introduced yet and will be described in the next section. In most powerful, general-purpose microprocessors, the stack is implemented in "software," i.e., within the memory. In order to keep track of the top of this stack within the memory, a 16-bit register is dedicated to the *stack pointer* or *SP*. The SP contains the address of the top of the stack within the memory. It will be shown that the stack is indispensable for interrupts and for subroutines.

top of the stack (two in the case of the Z80). The *pull* operation consists of removing one element from the stack. In the case of a microprocessor, it is the *accumulator* that will be deposited on top of the stack. The *pop* will result in a transfer of the top element of the stack into the accumulator. Other specialized instructions may exist to transfer the top of the stack between other specialized registers, such as the status register. The Z80 is more versatile than most in this respect.

The availability of a stack is required to implement three programming facilities within the computer system: subroutines, interrupts, and temporary data storage. The role of the stack during subroutines will be explained in Chapter 3 (Basic Programming Techniques). The role of the stack during interrupts will be explained in Chapter 6 (Input/Output Techniques). Finally, the role of the stack in saving data at high speed will be explained during specific application programs.

We will simply assume at

The Instruction Execution Cycle

Let us refer now to Figure 2.6. The microprocessor unit appears on the left, and the memory appears on the right. The memory chip may be a ROM or a RAM, or any other chip which happens to contain memory. The memory is used to store instructions and data. Here, we will fetch one instruction from the memory to illustrate the role of the program counter. We assume that the program counter has valid contents. It now holds a 16-bit address which is the address of the next instruction to fetch in the memory. Every processor proceeds in three cycles:

- 1—fetch the next instruction
- 2—decode the instruction
- 3—execute the instruction

Fetch

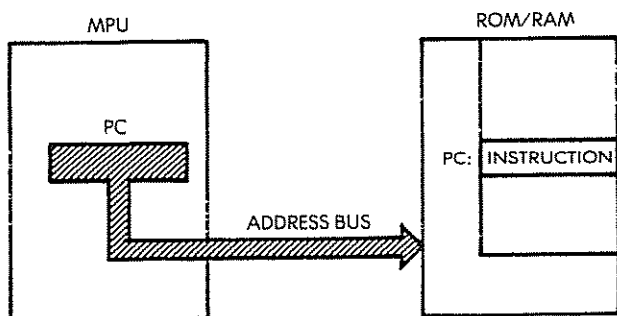
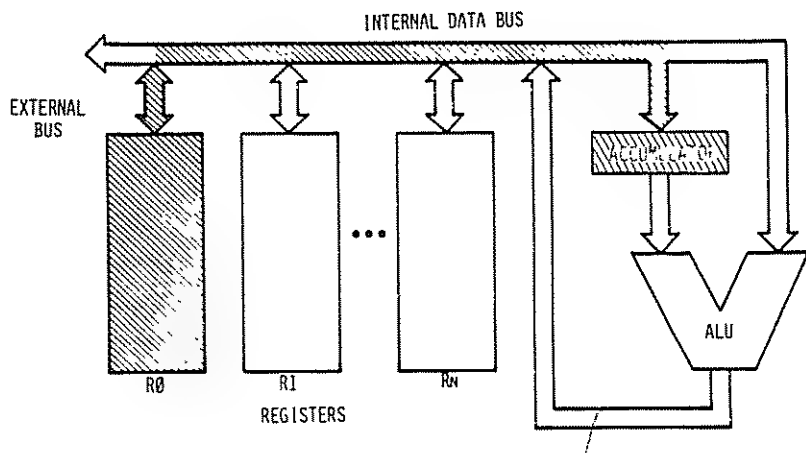


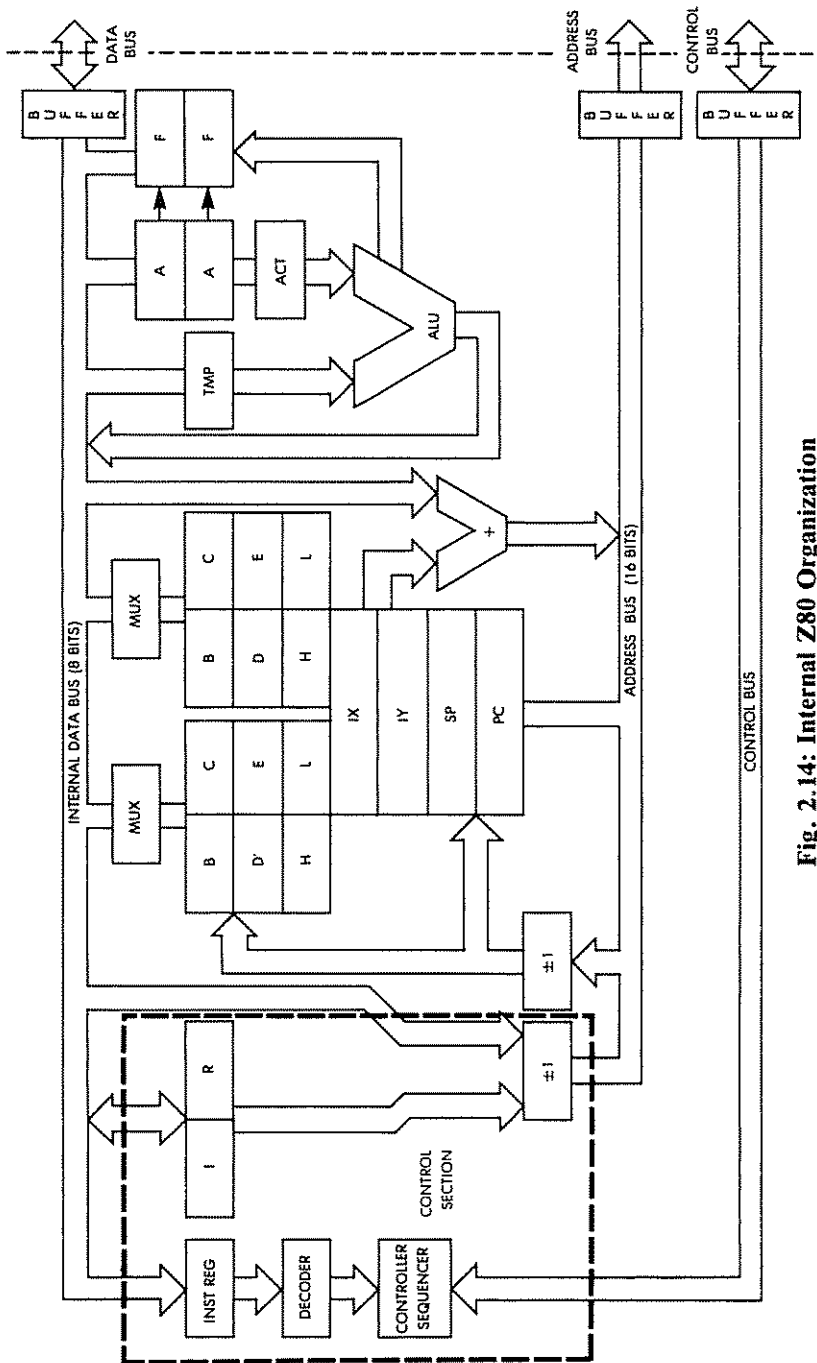
Fig. 2.6: Fetching an Instruction from the Memory

Dec

PROGRAMMING THE Z80

successive bytes of an instruction as well as to fetch successive instructions themselves. The program counter, together with its incrementer, provides an automatic mechanism for pointing to successive memory locations.





been chosen in order to maintain compatibility with the binary representation in this case. It is naturally arbitrary.

Exercise 2.1: Write below the binary code which will transfer the contents of register C into register B. Consult Fig. 2.16 for the codes corresponding to C and B.

Another simple example of a one-word instruction is:

ADD A, r

This instruction will result in adding the contents of a specified register (r) to the accumulator (A). Symbolically, this operation may be represented by: $A = A + r$. It can be verified in Chapter 4 that the binary representation of this instruction is:

The FETCH Phase

The FETCH phase of an instruction is implemented during the first three states of machine cycle M1; they are called T1, T2, and T3. These three states are common to all instructions of the microprocessor, as all instructions must be fetched prior to execution. The FETCH mechanism is the following:

T1 : PC OUT

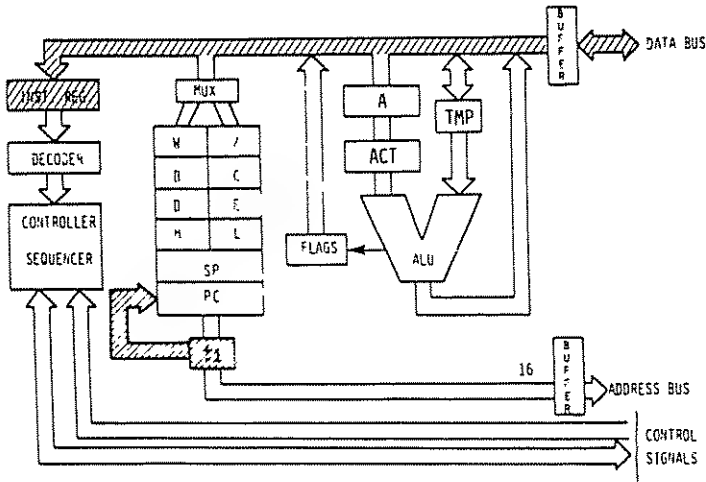
The first step is to present the address of the next instruction to the memory. This address is contained in the program counter (PC). As the first step of any instruction fetch, the contents of the PC are placed on the address bus (see Figure 2.17). At this point, an address is presented to the memory, and the memory address decoders will decode this address in order to select the appropriate location within the memory.

put pins of the memory, which are connected to the data bus. It is standard computer design to use the memory read time to perform an operation within the microprocessor. This operation is the incrementation of the program counter:

$$T2 : PC = PC + 1$$

While the memory is reading, the contents of the PC are incremented by 1 (see Figure 2.18). At the end of state T2, the contents of the memory are available and can be transferred within the microprocessor:

T3 INST into IR



During T5: (TMP) ► DDD.

The contents of TMP are deposited into D. This is shown in Figure 2.22.

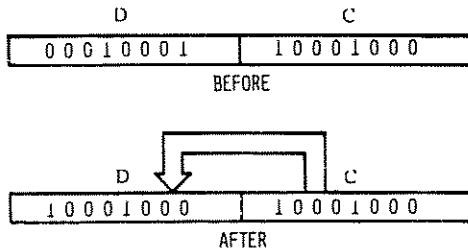


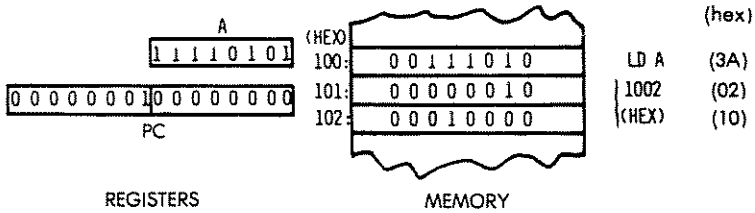
Fig. 2

Question: *What is the apparent execution time (to the programmer) for this instruction? Using a 2.5 Mhz clock, is it 3.6 us? 2.8 us?*

Another more complex instruction will now be examined which is a direct-memory addressing instruction using two invisible W and Z registers:

LD A,(nn)

The opcode is 00111010. The 8080 equivalent is LDA addr. As usual, states T1, T2, T3 of M1 will be used to fetch the instruction from the memory. T4 is used, but no visible result can be described. During state T4, the instruction is in fact

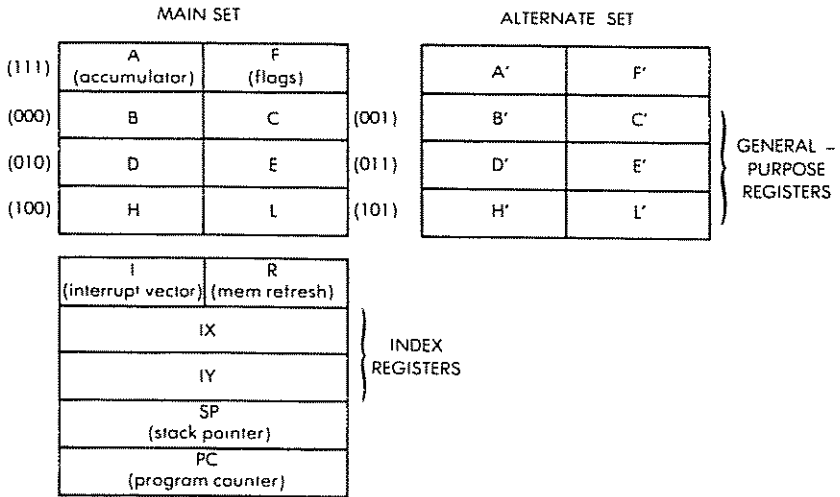


Understanding this point is crucial to the understanding of efficient execution of instructions within the microprocessor.

Question: (For the alert and informed reader only). What happens in the case of an interrupt at the end of M3? (If instruction execution is suspended at this point, the program counter points to the instruction following the jump, and the jump address, contained in W and Z, will be lost.)

The answer is left as an interesting exercise for the alert reader.

The detailed descriptions we have presented for the execution of typical instructions should clarify the role of the registers and of the internal buses. A second reading of the preceding section may help in gaining a detailed understanding of the internal operation of the Z80.



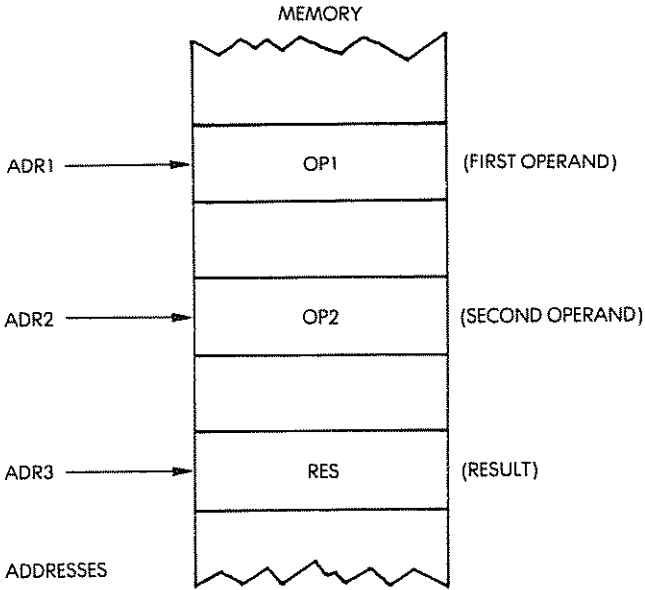


Fig.

